THE CODE PROJECT™
Your Development Resource

5,065,471 members and growing!
(11,044 online)

Email ______ Password ______ Sign in
Join ☑ Remember me? Lost your password?

Home | Other Technologies: | Java | LAMP | Help! | Articles | Message Boards | Lounge

» »

By

Posted:
Updated:
Views:

Note: This is an unedited reader contribution

Announcements

- Monthly Competition
- We're Hiring!
- Add Silverlight to your articles

Services

- Code Project Coffee
- Free Magazines

Chapters

- Desktop Development
- Web Development
- Enterprise Systems
- Multimedia
- Database
- Platforms, Frameworks & Libraries
- Languages
- General Programming
- Graphics / Design
- Development Lifecycle
- General Reading
- Third Party Products

Feature Zones

- Installshield 2008
- IBM DeveloperWorks

Search ______ Articles Go!
Advanced Search
Sitemap | Add to IE Search

Print | Report Article | Discuss | Send to a friend

- Download source files - 2 Kb

Introduction and Motivations

I am not a great lover of Multithreaded applications development, but at the end I come up with writing more multithreaded applications than single threaded ones. Though they are little bit difficult to manage, sometimes they provide us unique opportunity to solve a particular problem, provided your application is not CPU intensive. if your application is CPU intensive then multithreading will not give you its stated performance.

Every application is having its own thread creation policies for example

- Application may create a thread pool and waits for the requests to be serviced.
- Application may create thread on the fly when it needs to service some request.

In case 1 you need to have some mechanism to make a thread wait for some kind of signal. The signal may be generated in case when you need the service of any particular thread.

Using WIN32 Kernel Objects

WIN32 provides kernel objects like WIN32 Events to handle this kind of situations. you can use one of the WaitFor.. APIs to wait for a particular event inside a thread. The idea behind waiting for kernel objects is that it won't eats up all of the available CPU while waiting, otherwise you could have busy wait. Look at the following code.

```
#include <windows.h>
#include <iostream>
```

```
using namespace std;

HANDLE        hEvent;

DWORD    WINAPI ThrdFunc ( LPVOID n )
{
    int        TNumber    =    (int) n;
    WaitForSingleObject ( hEvent , INFINITE );
    cout<<"Event gets signaled...."<<endl;
    return 0;
}

int    main()
{
    HANDLE        hThrd;
    DWORD        Id;
    //    Create a Event
    hEvent    =    CreateEvent ( NULL , false , false , "Event 10234" );

    hThrd    =    CreateThread ( NULL , 0 ,
       (LPTHREAD_START_ROUTINE)ThrdFunc ,
       (LPVOID)0 , 0 , &Id );
    if ( !hThrd )
    {
        cout<<"Error Creating Thread ..."<<endl;
        return -1;
    }
    //    Wait For some time before giving out any signal
    Sleep ( 2000 );
    SetEvent ( hEvent );
    //    Wait for the thread to gets killed
    WaitForSingleObject ( hThrd , INFINITE );
    //    Close the handler
    CloseHandle ( hEvent );
    CloseHandle ( hThrd );
    return 0;
}
```

The code above shows that a thread is being created and it is waiting for an auto reset event. The event is signaled by the main thread after 2 seconds. once the waiting thread gets the signal it continues its execution and exit. You can achieve the same functionality using WIN32 Message queues.

Using WIN32 Message Queues

WIN32 message queues have certain advantages as compared to event based mechanism. As we will see you can send different messages to a single thread. The following code uses `PostThreadMessage` WIN32 API to post specific message to a particular thread.

Advantage Message Queue over Events

- You are sure that you are posting message to which thread, while in the case of events you can't guarantee which thread will get released if more than one threads are waiting on same event.
- You can send a range of user defined messages to a thread, if you want the thread to behave differently. However in case of events you can just signal

the events, no other information can be given.

Using Message queue in Multithreaded application

Message queues can be created even in console applications. you don't need a windows handle for it. Message queues are created in WIN32 console application as soon as you call message extractor functions like GetMessage and PeekMessage.

In my code below I have used GetMessage. PeekMessage is commented. The only harm in using PeekMessage is that PeekMessage doesn't waits for the message to come into the queue and will eats up all the available CPU which is undesirable in many cases. Here is the code

```
//    User Defines Messages
#define        THRD_MESSAGE_SOMEWORK        WM_USER + 1
#define        THRD_MESSAGE_EXIT            WM_USER + 2
//    Application Specific Preprocessor definitions
#define        NUM_THREADS                    2

DWORD WINAPI    ThrdFunc ( LPVOID n )
{
    int    TNumber    =    ( int )n;
    //    Here we will wait for the messages
    while ( 1 )
    {
        MSG    msg;
        //BOOL    MsgReturn  =  PeekMessage ( &msg , NULL ,
        //    THRD_MESSAGE_SOMEWORK , THRD_MESSAGE_EXIT , PM_REMOVE );
        BOOL    MsgReturn    =    GetMessage ( &msg , NULL ,
            THRD_MESSAGE_SOMEWORK , THRD_MESSAGE_EXIT );

        if ( MsgReturn )
        {
            switch ( msg.message )
            {
            case THRD_MESSAGE_SOMEWORK:
                cout<<"Working Message.... for Thread Number "
                   <<TNumber<<endl;
                break;
            case THRD_MESSAGE_EXIT:
                cout<<"Exiting Message... for Thread Number "
                   <<TNumber<<endl;
                return 0;
            }
        }
    }

    return 0;
}

int main()
{
    HANDLE    hThrd [ NUM_THREADS ];
    DWORD    Id [ NUM_THREADS ];
    short LoopCounter = 0;
    //    Create all the threads
    for ( ; LoopCounter < NUM_THREADS ; LoopCounter ++ )
    {
        hThrd [ LoopCounter ] = CreateThread ( NULL , 0 ,
            (LPTHREAD_START_ROUTINE)ThrdFunc ,
            (LPVOID)LoopCounter , 0, &Id [ LoopCounter ] );
        if ( !hThrd )
        {
            cout<<"Error Creating Threads,,,,.exiting"<<endl;
            return -1;
        }
        Sleep ( 100 );
    }
    Sleep ( 10000 );
    //    Send Working Message to all Threads
    for ( LoopCounter = 0; LoopCounter < NUM_THREADS ; LoopCounter ++ )
    {
        PostThreadMessage ( Id [ LoopCounter ] ,
            THRD_MESSAGE_SOMEWORK , 0 , 0 );
        Sleep ( 100 );
    }
    //    Sleep againg for 1 seconds and send exit message to all threads
    Sleep ( 1000 );
    for ( LoopCounter = 0; LoopCounter < NUM_THREADS ; LoopCounter ++ )
    {
        PostThreadMessage ( Id [ LoopCounter ] , THRD_MESSAGE_EXIT, 0 , 0 );
        Sleep ( 100 );
    }
    //    Wait for all threads to exit
    WaitForMultipleObjects ( NUM_THREADS, hThrd , true , INFINITE );
    //    Close All handles
    for ( LoopCounter = 0; LoopCounter < NUM_THREADS ; LoopCounter ++ )
        CloseHandler ( hThrd [ LoopCounter ] );

    return 0;
}
```

In the code above we created two user defined messages. Each thread waits for the particular message. The message is send to all threads from the main thread. The main thread waits for all threads to exit before exiting itself.

History

- Last updated on Feb 26, 2004

License

Article Top	Sign Up to vote for this article	***Rate this Article for us!***	*Poor* ○ ○ ○ ○ ○	*Excellent* Vote

Discussions and Feedback

Visit to post and view comments on this article, or click **here** to get a print view with messages.